Design Rationale Revision 3

Object-Oriented Programming Principle implementations:

New Class Implementations:

E N E M Y

Enemy class is an abstract class extending the Actor class, representing the children classes and acts as a base code for its child classes: Undead, Skeletons and Lord of Cinder. To ensure that an instance of the Enemy class is not instantiated. The death of Enemies will drop Souls, the currency of the game. Each Enemy will have the base Souls count of 50 for Undead, 250 for Skeletons and 5000 for Lord of Cinder. These values are stored as a constructor parameter for creating instances of each enemy. The main objective is for the Player to pick up the Souls that are dropped by Enemies after they are killed and added into the Player's inventory and most importantly, the attacks between the Player and Enemy. The damages dealt by the Enemy can be stored in this class to prevent duplicate code in the Enemy children classes, instead, the children classes can just override this method. This is again an example of the implementation of the DRY principle as well as avoiding the excessive use of literals as it is not hard-coded, making the maintenance of the code smoother and easier in the future.

Enemy class is created to follow the Reduce Dependency Principle as the Actor class is initially inherited by the Player, Undead, Skeleton, LordOfCinder and Ground class. Hence, the Enemy class will inherit the Actor class instead, which is then inherited by its children classes like the Undead and EnemyWithWeapon (inherited by Skeletons and Yhrom) reducing dependency on the Actor class.

Enemy class will also be inclusive of an Arraylist of the Actor data type, keeping track of if the Actor is a Player or an Enemy is close together to determine the behaviour. This attribute will be looped through to determine if Enemy will use AttackAction() if the Player is in the target range. Thus, also reducing the use of duplicate code and utilising the DRY principle.

S K E L E T O N

Skeleton class will extend the EnemyWithWeapon class as it deals damage and displays phrases like "thwacks" or "punches". Each skeleton will also deal base damage to the player signifying how it will override the method indirectly inherited from the Enemy class. This class will implement a spawnEnemies() method which spawns 4 to 12 skeletons on the map. It is given the range of Skeletons that need to be spawned on the map and utilises WanderBahvaiour() to place the Skeleton in a random location of the map. As well as using the revived() method Skeletons class will inherit the parent constructor and will have these values as parameters: 100 hitpoints, 250Souls and a Weapon.

B O N F I R E

Bonfire is an extension of Ground. Overriding the canActorEnter() method and setting the return boolean value to False if the Actor is an Enemy. It is also responsible for recharging the Player' number of Estus Flask if it is less than 3. A new method rechargeEstusFlask() can be done in this class, where the boolean method checks if the Player has less than 3 Estus Flasks, if it returns True, then it is recharged to 3 Estus Flaks; else, do nothing and return a boolean value of False. This class is also responsible for resetting the Enemy positions once the Player dies. The vendor extends this class as the Vendor's actions are done on this terrain, Bonfire.

V E N D O R

This class will hold the methods in order for the Player to trade Souls for upgrades or Weapons. Vendor implements Souls as Souls is the object that interacts between Player and Vendor for the upgrades or weapon purchases. As the Player purchases a Weapon, the SwapWeaponAction which is a part of PickUpItemAction() is then called. This occurs as the Player can only wield one weapon at a time.

C E M E T E R Y

Cemetery class extends Ground class as the class represents the different Ground-type. The cemetery has a special trait where the Undead are spawned in this terrain type. A requirement for this class would also be that there will be at least 5 Cemeteries in the GameMap. Having this being a requirement, a method implemented is required to check the number of Cemeteries within the GameMap, if it is less than 5 the newGround() method in FancyGroundFectory class will be called to create more Cemetery Grounds. This is again an example of the DRY principle where codes is not repeated as the newGround() is called instead of making a new method specifically for creating new Grounds in the GameMap.

P U R C H A S E A C T I O N

A class extending Action in which the player is allowed to interact with the Vendor to purchase weapons using souls. Having both MeleeWeapon having a similar abstract which simplifies the passing of the instance using the interface. This refers to the Dependency Inversion Principle (DIP) where high-level modules should not depend on low-level modules which refers to how Action does not depend on PurchaseAction but rather separating the different Action possible from Action class. The Action is then added to Vendor. Class fitting into the SRP where the class is only responsible for purchases of weapons in game.

U P G R A D E A C T I O N

Similar to PurchaseAction class, this is also an extension of Action in which the player is allowed to upgrade HP by trading souls with the Vendor. This method just varies from the lack of use of accessors, but rather Player mutators. In which it checks if the player has a sucessfulTransaction utilising subtractSouls() and increase HP if the return value is True. Class fitting into the SRP where the class is only responsible for trading of souls to increase player HP.

C H E S T

Fulling the need of DRY principle, where a Chest is just a Ground object that Player can interact with. Being declared as a Ground-type, with all the base functionalities like if Player can enter. This is then extended by ChestAction to perform the various Chest scenarios of turning into Mimic or TokenOfSouls. Also, a demonstration of the Open/Closed principle, open to be extended by ChestAction but closed for modification.

C H E S T A C T I O N

This class is made to fulfil the SRP where this class is responsible for actions Chest can have which is the scenarios Chest can have which is turning into Mimic or drop a pile of TokenOfSouls. Tasked with the making of the Enemy Mimic based on a random boolean value. Instead of relying on this task under Chest, a new class benign ChestAction is made.

F O G D O O R

An extension of Ground which fits under the DRY principle in which a Ground-type is made with the same functions as the parent class Ground. With a couple of overridden methods so Player can interact with FogDoor and returning false for Enemies where they don't have the Enum Type Player. Responsible for making FogDoor instance under Application where it is then able to access both GameMap instances. As well as fulfilling the SRP where the class has a Single responsibility which is to move the Player between maps.

M I M I C

Extended from Enemy backed up by SRP where the class is responsible for the instance of Mimic when Mimic is made. When Mimic is made, it will attack the Player with an Intrinsic Weapon. This class is made to be responsible for the actions of Mimic like attacking. But Mimic does not wander and etc.

ALDRICH DEVOURER

This class is similar to the Yhorm enemy boss class, and as such, also extends from the main parent Enemy class, alongside implementing both the Souls and Resettable Interfaces. By extending from the enemy classes and implementing the listed interfaces, the DRY principle is being fulfilled. In addition to this, the boss class overrides the PlayTurn method, further exhibiting the use of the DRY principle. Additionally, by implementing the Resettable interface, the program is able to easily reset the boss hit points and move the boss to its starting position.

DARKMOON LONGBOW

The DarkmoonLongbow class inherits from the weaponItem engine class and creates a special weapon for Aldrich the Devourer. The class is able to inherit all the WeaponItem and Item parent class's methods and variables exhibiting the use of encapsulation and the DRY principle, as the relevant data and methods are inherited rather than hard-coded multiple times.